Dariusz BURAK, Piotr BŁASZYŃSKI
WEST POMERANIAN UNIVERSITY OF TECHNOLOGY,
DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

Parallelization of the Camellia Encryption Algorithm

Dr inż. Dariusz BURAK

Received the PhD degree in computer science in 2007 from Szczecin University of Technology. He is an assistant professor of the computer science at the West Pomeranian University of Technology. His current research interests are focused on cryptography and compiler optimizations.

e-mail: dburak@wi.zut.zut.edu.pl

Dr inż. Piotr BŁASZYŃSKI

Received the PhD degree in computer science in 2004 from Szczecin University of Technology. He is an assistant professor of the computer science at the West Pomeranian University of Technology. His current research interests are focused on embedded systems and advanced compilers techniques.

e-mail: pblaszynski@wi.zut.zut.edu.pl

Abstract

A parallelization process of the Camellia encryption algorithm along with the description of exploited parallelization tools is presented. The data dependency analysis of loops and the loop transformations were applied in order to parallelize the sequential algorithm. The OpenMP standard was chosen for representing parallelism of the cipher. Speed-up measurements for a parallel program are presented.

Keywords: Camellia encryption algorithm, parallelization, data dependency analysis, OpenMP.

Zrównoleglenie algorytmu szyfrowania Camellia

Streszczenie

W artykule przedstawiono proces zrównoleglenia japońskiego standardu szyfrowania danych - blokowego algorytmu szyfrowania Camellia, bazującego na sieci Feistela, pracującego w trybie pracy ECB. Krótko opisano wykorzystane do tego celu narzędzia programowe: program Petit, który służy do analizy istniejących zależności danych w pętlach programowych oraz OpenMP API. W celu zrównoleglenia algorytmu sekwencyjnego zastosowano analizę zależności danych oraz dokonano przekształceń pętli programowych w celu wyeliminowania istniejących zależności pętli blokujących proces ich zrównoleglenia. Do prezentacji równoległości szyfru wybrano język C oraz standard OpenMP. Załączono również wyniki pomiarów przyspieszenia pracy programu równoległego oraz najbardziej czasochłonnych pętli, które są odpowiedzialne za proces szyfrowania oraz deszyfrowania danych dla dwóch, czterech, ośmiu oraz szesnastu procesorów oraz dla dwóch, czterech, ośmiu oraz szesnastu wątków utworzonych z zastosowaniem kompilatora Intel® C++ w wersji.11.0 zawierającego OpenMP API w wersji 3.0. Najbardziej czasochłonne pętle zostały w pełni zrównoleglone, natomiast przyspieszenie pracy całego programu, zgodnie z prawem Amdahla jest zredukowane z uwagi na występowanie w kodzie programu części sekwencyjnej, zawierającej sekwencyjne operacje wejścia- wyjścia służące do odczytu danych z pliku, oraz zapisu danych do pliku. Wyniki zrównoleglenia opisane w artykule mogą być pomocne do implementacji sprzętowych algorytmu Camellia.

Słowa kluczowe: algorytm szyfrowania Camellia, zrównoleglenie, analiza zależności danych, OpenMP.

1. Introduction

One of the most important functional features of cryptographic algorithms is a cipher speed, and thus even a little difference of speed may cause the choice of the faster cipher by the user. Therefore, it is so important to enable the use of Shared Memory Parallel Computers for cryptographic algorithms processing. The paper describes a software approach based on the transformations of a source code written in C language representing the sequential Camellia encryption algorithm. However, the creation of parallel algorithms is connected with the current world tendency towards the hardware implementation of cryptographic algorithms, because we also need parallel algorithms in this case. The major purpose of this paper is to present a parallelization process of the Camellia encryption algorithm along with the description of exploited parallelization tools. The paper is organized as follows. In section 2, the Camellia encryption algorithm is briefly described. Section 3 contains a brief description of data dependency analysis and the Petit program. Section 4 discusses parallelization process of the Camellia encryption algorithm. Section 5 shows experimental results regarding the application efficiency of the parallelized Camellia algorithm. Conclusion remarks are given in Section 6.

2. The Camellia encryption algorithm

Camellia is a block cipher developed by Nippon Telegraph & Telephone Corporation and Mitsubishi Electric Corporation in 2000 [1]. Camellia was chosen as a recommended algorithm by the NESSIE (New European Schemes for Signatures, Integrity and Encryption) project in 2003 [2] and also was certified as the IETF (Internet Engineering Task Force) standard cipher for XML security URIs [3], SSL/TLS cipher suites [4] and IPsec in 2005 [5].

Camellia is based on the Feistel network that operates on 128-bit data blocks with a 128-bit, 192-bit or 256-bit key and with 22-rounds (28-rounds for 192-bit and 256-bit key) data processing composed of three main parts: an 18-rounds Feistel structure (24-rounds Feistel structure for 192-bit and 256-bit key), two FL and FL-1 functions and two input/output whitenings.

In Camellia, four types of S-boxes are applied and each one consists of a multiplicative inversion and affine transformations. A linear 64-bit permutation follows the nonlinear substitution of S-boxes. The FL and FL-1 functions inserted every 6 rounds are used to provide non-regularity between the rounds. These two functions are similarly constructed by logical operations including AND, OR, XOR, and rotations.

The decryption process is performed in the same way as encryption one except the subkeys should be used in a reverse order. The key scheduler shares part of the process with encryption and has additional rotations of the subkeys.

Camellia encryption algorithm is the international standard cipher-alternative to the Advanced Encryption Standard (AES).

Camellia essential patents can be used at no charge by any Camellia user [6].

NTT publishes NTT's open source codes of Camellia free of charge through multiple open source software licences (GPL, LGPL, BSD, MPL, and OpenSSL) [7].

3. Data Dependency Analysis

In order to parallelize loops contained in the Camellia encryption algorithm data dependency analysis using Petit program was performed.

Developed at the University of Maryland under the Omega Project and freely available for both DOS and UNIX systems, Petit program is a research tool for analyzing array data dependences [8]. Petit operates on programs in a simple Fortran-like language and provides indispensable information about the following data dependences that occur in the analyzing loop: value- based dependences, memory- based dependences, loop carried dependences, data flow dependences, antidependences and output dependences.

There are the following three types of data dependences blocking parallelism in "for" loops [9, 10]:

1. Data flow dependence indicates that a write-before-read ordering must be satisfied for parallel computing.
2. Antidependence indicates that a read-before-write ordering should not be violated for parallel computing.
3. Output dependence indicates a write-before-write ordering for parallel processing. .

There is also another type of dependence called control dependence. In this case, value of variable (in S2) depends on the flow of control (in S1) like in the following example:

```
for(int i=0; i<n; i++) {
S1: if(x!=0)
S2: y = 1.0/x;
}.
```

All of the above loops cannot be executed in parallel in such a form, because results generated by the loops would be not the same as whose yielded with sequential loops. Thus, it is necessary to transform these loops so as to eliminate such dependences.

4. Parallelization process of the Camellia encryption algorithm

In order to present parallelized source code of the Camellia encryption algorithm the OpenMP API was applied.

The OpenMP Application Program Interface (API) supports multi-platform shared memory parallel programming in C/C++ and Fortran on all architectures including Unix and Windows NT platforms. OpenMP is a collection of compiler directives, library routines and environment variables that can be used to specify shared memory parallelism. OpenMP directives extend a sequential programming language with Single Program Multiple Data (SPMD) constructs, work-sharing constructs and synchronization constructs and enable to operate on private data. An OpenMP program begins execution as a single task called the master thread. When parallel construct is encountered, the master thread creates a team of threads. The statements within parallel construct are executed in parallel by each thread in the team. At the end of the parallel construct, all threads in the team are synchronized. Then only the master thread continues execution until the next parallel construct will be encountered [11].

In order to parallelize the Camellia encryption algorithm in the ECB mode, it was used sequential Camellia algorithm module written in ANSI C language- version M1.02 derived from the original code furnished in open source by NTT and Mitsubishi Electric Corporation [7]. This choice makes possible to perform efficient parallelization in view of some advantageous features of that source code (a high clarity, enclosing the most of computations in iterative loops, a little number of used functions). In order to enable enciphering and deciphering the whichever number of data blocks, we have created two new functions, the Camellia_enc() for the encryption process (based on Camellia_Encrypt()) and the Camellia_dec() for the decryption process (based on Camellia_Decrypt()), by analogy with similar functions included in the C source code of the cryptographic algorithms (DES- the des_enc(), the des_dec(), LOKI91- the loki_enc(), the loki_dec, IDEA- the idea_enc(), the idea_dec(), etc.) presented in [12].

The process of the Camellia encryption algorithm parallelization can be divided into the following stages:

1. Finding the most time-consuming functions of the Camellia encryption algorithm;
2. Making preliminary transformations of the most time-consuming loops;
3. Data dependency analysis of the most time-consuming loops using Petit program;
4. Removal of data and control dependences (when it is possible);
5. Constructing parallel loops (in accordance with the OpenMP standard);
6. Verification of a parallelized source code.

It has been carried out experiments with the sequential Camellia encryption algorithm that encrypts and then decrypts 3 megabytes plaintext in order to find the most time-consuming functions including no I/O functions. It has been discovered that such functions are included in the Camellia_enc() and in the Camellia_dec() thus their parallelization is critical for reducing the total time of the algorithm execution.. Taking into account the strong similarity of both loops we show only Camellia_enc() function:

```
void Camellia_enc( const int n, const Byte *p, const Byte *e,
Byte *c, int blocks) {
int i, ii;
for (ii=0; ii<blocks; ii++) {
XorBlock( p+16*ii, e+0, c+16*ii);
for( i=0; i<3; i++ ){
Camellia_Feistel( c+0+16*ii, e+16+(i<<4), c+8+16*ii);
Camellia_Feistel( c+8+16*ii, e+24+(i<<4), c+0+16*ii);
                                }
Camellia_FLlayer( c+16*ii, e+64, e+72);
for( i=0; i<3; i++ ){
Camellia_Feistel( c+0+16*ii, e+80+(i<<4), c+8+16*ii);
Camellia_Feistel( c+8+16*ii, e+88+(i<<4), c+0+16*ii);
}
Camellia_FLlayer( c+16*ii, e+128, e+136);
for( i=0; i<3; i++ ){
Camellia_Feistel( c+0+16*ii, e+144+(i<<4), c+8+16*ii);
Camellia_Feistel( c+8+16*ii, e+152+(i<<4), c+0+16*ii);
}
if( n == 128 ){
SwapHalf( c+16*ii);
XorBlock( c+16*ii, e+192, c+16*ii);
}
else {
Camellia_FLlayer( c+16*ii, e+192, e+200 );
for( i=0; i<3; i++ ){
Camellia_Feistel( c+0+16*ii, e+208+(i<<4), c+8+16*ii);
Camellia_Feistel( c+8+16*ii, e+216+(i<<4), c+0+16*ii);
}
SwapHalf( c+16*ii);
XorBlock( c+16*ii, e+256, c+16*ii);
}
}
}.
```

The parallelization process is presented only for the loops included in function Camellia_enc() (however, this analysis is valid also for the second one).

We have chosen to make parallelization of the outer loop (indexed by the variable ii) in order to parallelize the source code included in this function and exploit the coarse- grained data parallelism available at the loop level. Among others, four not perfectly nested loops (indexed by the variable i) are included in the outer loop. Therefore, we exploit the coarse- grained data parallelism available at the outer loop level.

To remove data dependences existing in the source code we have to make the privatization of the following variables: i, ii.

The source code of the outer loop is suitable to apply the following OpenMP API constructs [13, 14]:
- parallel region construct ("parallel" directive)
- work-sharing construct ("for" directive)- all the iterations of the associated loop can be executed in parallel in this case.

Thus the Camellia_enc() function with the parallelized most time-consuming loop has the following form (in accordance with the OpenMP API):

```
void Camellia_enc( const int n, const Byte *p, const Byte *e,
Byte *c, int blocks) {
int i, ii;
#pragma omp parallel private(i,ii)
#pragma omp for
for (ii=0; ii<blocks; ii++) {
…//body of the loop
}.
```

5. Speed-up measurements

In order to study the efficiency of the parallel code the computer with the following features was used:
- 8 x Quad Core Intel Xeon Processor Model E7310,
- the openSuse 11.1 operating system,
- the Intel® C++ Compiler ver.11.0 (that supports the OpenMP 3.0).

The results received for the plaintext of the size about 10 megabytes are shown in Table 1.

The total running time of the Camellia encryption algorithm consists of the following time-consuming operations:
1. Data reading from an input file;
2. Data encryption;
3. Data decryption;
4. Data writing to an output file (both encrypted and decrypted text).

Tab. 1. Speed-up measurements of the Camellia encryption algorithm in the ECB mode
Tab. 1. Wyniki pomiaroów przyspieszenia pracy algorytmu szyfrowania Camellia w trybie pracy ECB

No. of processors	No. of threads	Speed-up		
		Encryption	Decryption	Total
1	1	1.0	1.0	1.0
2	2	1.8	2.0	1.4
4	4	3.2	3.7	1.8
8	8	3.8	4.2	2.1
16	16	3.7	4.1	2

The total speed-up of the parallelized Camellia encryption algorithm depends considerably on the two major factors:
- whether the most time-consuming loops are parallelizable
- the method of data reading and data writing.

The results confirm that the paralleled codes of the most time-consuming loops (placed in the Camellia_enc() and the Camellia_dec() functions) have sufficient efficiencies. The block method of reading data from an input file and writing data to an output file was used. The following C language functions and block sizes were applied:
- the fread() function (with the 32-bytes block for data reading),
- the fwrite() function (with the 512-bytes block for data writing),

(the optimal sizes of the blocks were chosen via the appropriate number of tests with various block sizes).

In accordance with Amdahl's Law [15, 16] the maximum speed-up of the whole Camellia algorithm is limited to 5.55, because the fraction of the code that cannot be parallelized is 0.180. This fraction is calculated as the quotient of the sum of the execution time of all unparallelizable operations divided by the execution time of the whole algorithm.

6. Conclusions

In this paper, the parallelization process of the Camellia encryption algorithm was described. This algorithm could be divided into parallelizable and unparallelizable parts. It was shown that the most time-consuming iterative loops (responsible for the data blocks encryption and the data blocks decryption) are fully parallelizable. In order to parallelize these loops it is necessary to make appropriate transformations of the source code of these loops (as was described in Section 4). The experiments carried out on the multiprocessor computer (with four- core processors) with one, two, four, eight and sixteen threads show that the application of the parallel Camellia encryption algorithm considerably boost the time of the data encryption and decryption processes. The rest of the time-consuming parts of code, contains I/O functions that are unparallelizable because the access to memory is, by its very nature, sequential. Hence, the total speed-up is less than that for the parallelizable part. The parallel Camellia encryption algorithm presented in this paper can be also helpful for hardware implementations. The hardware synthesis of the Camellia encryption algorithm will depend on the appropriate adjustment of the data transmission capacity and the computational power of hardware.

7. References

[1] Aoki K., Ichikawa T., Kanda M., Matsui M., Moriai S., Nakajima J., and Tokita, T.: Camellia: A 128-Bit Block Cipher Suitable for Multiple Platforms - Design and Analysis -, In Selected Areas in Cryptography, 7th Annual International Workshop, SAC 2000, August 2000, Proceedings, Lecture Notes in Computer Science 2012, pp.39-56, Springer-Verlag, 2001.
[2] The NESSIE project (New European Schemes for Signatures, Integrity and Encryption), http://www.cryptonessie.org
[3] IETF RFC 4051 Additional XML Security URIs, April 2005, http://www.ietf.org/rfc/rfc4051.txt
[4] IETF RFC 4132 Addition of Camellia Cipher Suites to Transport Layer Security (TLS), July 2005, http://www.ietf.org/rfc/rfc4132.txt
[5] IETF RFC 4312 The Camellia Cipher Algorithm and Its Use With IPsec, December 2005, http://www.ietf.org/rfc/rfc4312.txt
[6] http://info.isl.ntt.co.jp/crypt/eng/camellia/intro.html
[7] http://info.isl.ntt.co.jp/crypt/eng/camellia/source.html
[8] Kelly W., Maslov V., Pugh W., Rosser E., Shpeisman T., Wonnacott D.: New User Interface for Petit and Other Extensions. User Guide, 1996.
[9] Moldovan D.I.: Parallel Processing. From Applications to Systems, Morgan Kaufmann Publishers, Inc., 1993.
[10] Allen R., Kennedy K.: Optimizing compilers for modern architectures: A Dependence-based Approach, Morgan Kaufmann Publishers, Inc., 2001.
[11] Chandra R., Dagum L., Kohr D., Maydan D., McDonald J., Menon R.: Parallel Programming in OpenMP, Morgan Kaufmann Publishers, Inc., 2001.
[12] Schneier B.: Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition, John Wiley & Sons, 1995.
[13] Quinn M.J.: Parallel Programming in C with MPI and OpenMP, McGraw-Hill, 2004.
[14] OpenMP Application Program Interface, Version 3.0, May 2008.
[15] Amdahl G.M.: Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities, In AFIPS Conference Proceedings, 1967.
[16] Bielecki W.: Essentials of parallel and distributed computing, Informa, 2002.

Artykuł recenzowany